Ex. No: 07	**Triggers**
Date	3-10-2023

Objective:

To solve the given problems using triggers.

Software Required:

Oracle 10g

Description:

Triggers are similar to stored procedures. A trigger stored in the database can include SQL and PL/SQL or Java statements to run as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger.

Detailed Procedure:

The events that fire a trigger include the following:

DML statements that modify data in a table (INSERT, UPDATE, or DELETE) DDL statements System events such as startup, shutdown, and error messages User events such as logon and logoff A trigger has three basic parts:A triggering event or statement ,trigger restriction , trigger has three basic parts:riggering event or statement ,trigger restriction A trigger action

The types of triggers are:

BEFORE and AFTER Triggers

INSTEAD OF Triggers

Triggers on System Events and User Events

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for keeping the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should be created also in the tables of the taxes, vacations, and salaries.

Sample Input /Output

1. Create a trigger T1_sal that prints “Salary incremented “whenever there is a increase in salary and “salary decremented “whenever there is a decrease in salary in employees table

```
create or replace trigger t1_sal
after update of salary on employee
for each row
Begin
 if (:new.salary>:old.salary) then
        dbms_output.put_line('Salary Incremented');
 elsif (:new.salary<:old.salary) then
 dbms_output.put_line('Salary Decremented');
 end if;
 end;
```

2. Create a trigger T2 _error that raises an error whenever there is no commission for manager for new insertions

```
create or replace trigger t2_error
     after insert on emp
     for each row
     begin
     if(:new.comm is null) then
     raise_application_error(-20000,'commission not given');
     end if;
     end;
```

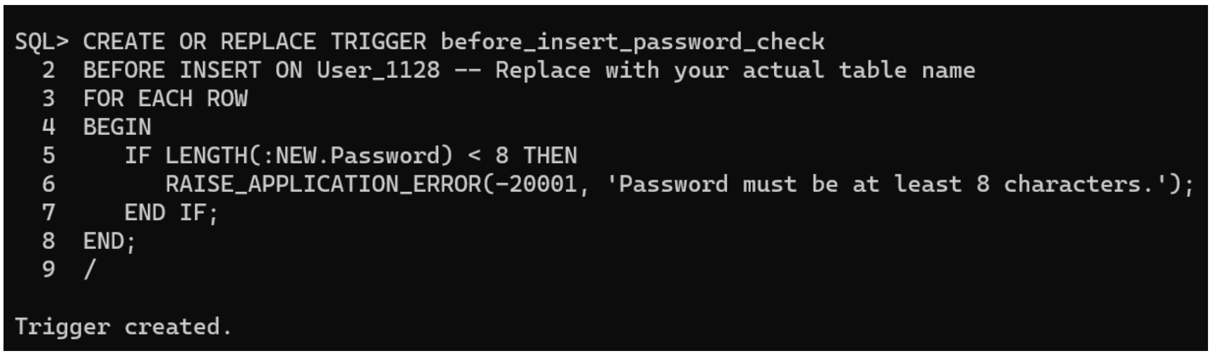
3. Create a trigger T3 _set_null to set the manager id null in the departments table whenever the manager id is deleted from the employees table.

```
 create or replace trigger t3_set_null
      after delete  on employee
      for each row
      begin
      update departments set manager_id=null where manager_id=:old.manager_id;
      end;
```

Question:

1. Create a BEFORE INSERT Trigger for the "User" Table that ensures that the passwords are at least 8 Characters.

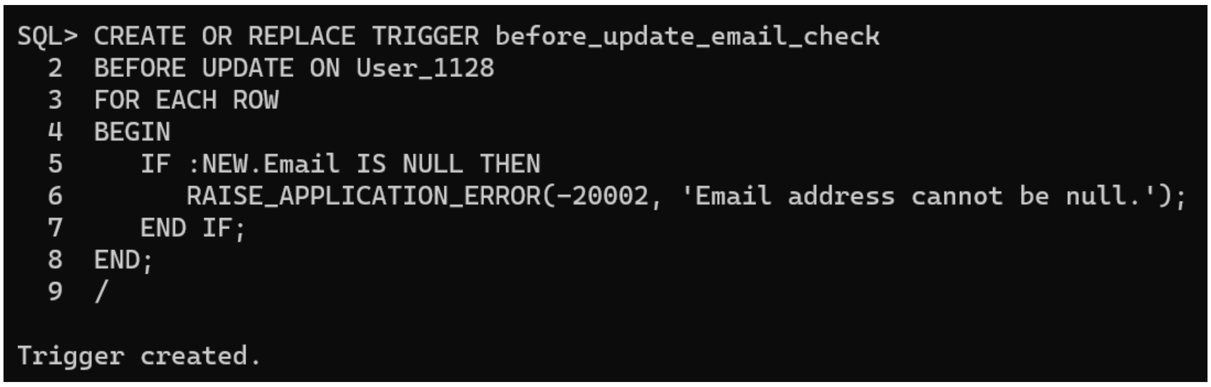
```
CREATE OR REPLACE TRIGGER before_insert_password_check

BEFORE INSERT ON User_1128

FOR EACH ROW

BEGIN

  IF LENGTH(:NEW.Password) < 8 THEN

    RAISE_APPLICATION_ERROR(-20001, 'Password must be at least 8 characters.');

  END IF;

END;

/
```



```
SQL> CREATE OR REPLACE TRIGGER before_insert_password_check
BEFORE INSERT ON User_1128 -- Replace with your actual table name
FOR EACH ROW
BEGIN
   IF LENGTH(:NEW.Password) < 8 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Password must be at least 8 characters.');
   END IF;
END;
/

Trigger created.
```

2. Create a BEFORE UPDATE Trigger for the "User" Table that does not allow email addresses to be null.

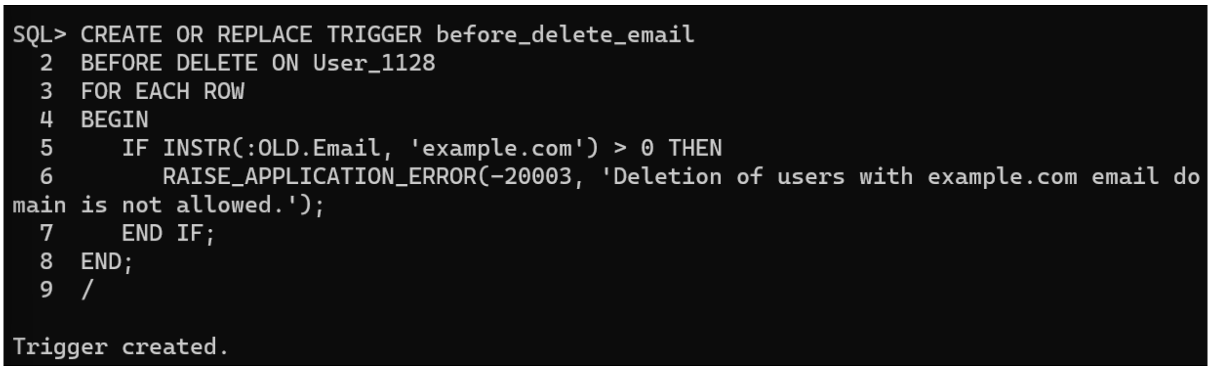
```
CREATE OR REPLACE TRIGGER before_update_email_check

BEFORE UPDATE ON User_1128

FOR EACH ROW

BEGIN

  IF :NEW.Email IS NULL THEN

    RAISE_APPLICATION_ERROR(-20002, 'Email address cannot be null.');

  END IF;

END;

/
```



```
SQL> CREATE OR REPLACE TRIGGER before_update_email_check
  2  BEFORE UPDATE ON User_1128
  3  FOR EACH ROW
  4  BEGIN
  5     IF :NEW.Email IS NULL THEN
  6        RAISE_APPLICATION_ERROR(-20002, 'Email address cannot be null.');
  7     END IF;
  8  END;
  9  /

Trigger created.
```

3. Create a BEFORE DELETE Trigger for the “User” Table that prevents the deletion of users with specific email domains (like "example.com").

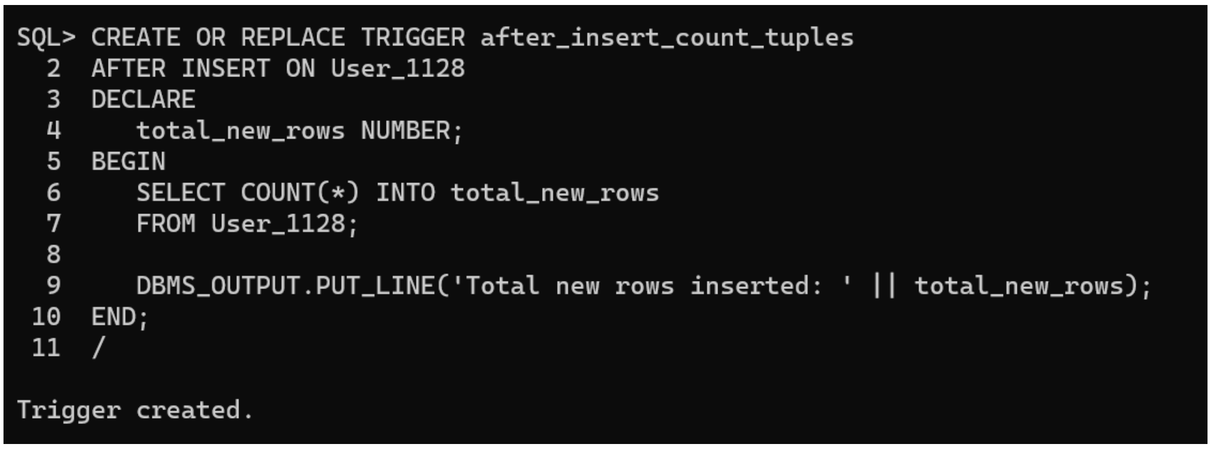
```
CREATE OR REPLACE TRIGGER before_delete_email
BEFORE DELETE ON User_1128
FOR EACH ROW
BEGIN
  IF INSTR(:OLD.Email, 'example.com') > 0 THEN
    RAISE_APPLICATION_ERROR(-20003, 'Deletion of users with example.com
email domain is not allowed.');
  END IF;
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER before_delete_email
  2  BEFORE DELETE ON User_1128
  3  FOR EACH ROW
  4  BEGIN
  5     IF INSTR(:OLD.Email, 'example.com') > 0 THEN
  6        RAISE_APPLICATION_ERROR(-20003, 'Deletion of users with example.com email do
main is not allowed.');
  7     END IF;
  8  END;
  9  /

Trigger created.
```

4. Write an AFTER INSERT trigger to count number of new tuples inserted using each

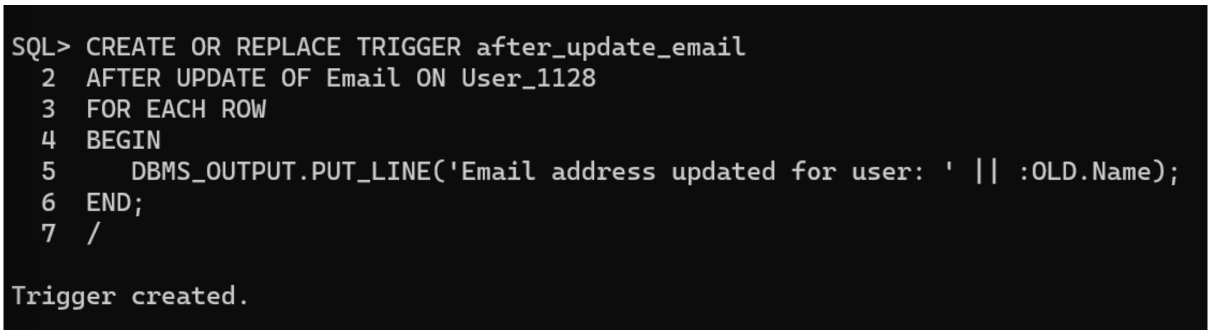
```
CREATE OR REPLACE TRIGGER after_insert_count_tuples
AFTER INSERT ON User_1128
DECLARE
  total_new_rows NUMBER;
BEGIN
  SELECT COUNT(*) INTO total_new_rows
  FROM User_1128;

  DBMS_OUTPUT.PUT_LINE('Total new rows inserted: ' || total_new_rows);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_insert_count_tuples
  2  AFTER INSERT ON User_1128
  3  DECLARE
  4     total_new_rows NUMBER;
  5  BEGIN
  6     SELECT COUNT(*) INTO total_new_rows
  7     FROM User_1128;
  8
  9     DBMS_OUTPUT.PUT_LINE('Total new rows inserted: ' || total_new_rows);
 10  END;
 11  /

Trigger created.
```

5. Create an AFTER UPDATE Trigger for the “User” Table that signals when a user's email is changed.

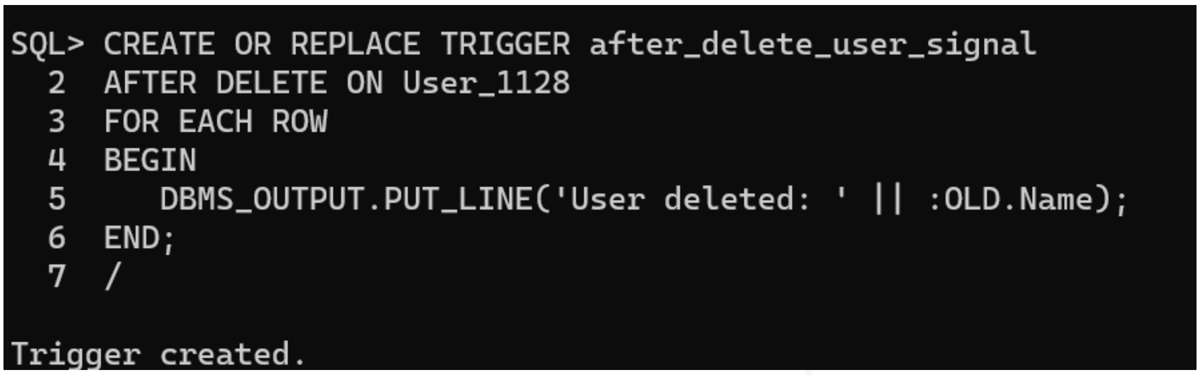
```
CREATE OR REPLACE TRIGGER after_update_email
AFTER UPDATE OF Email ON User_1128
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Email address updated for user: ' || :OLD.Name);
END;
    /
```



```
SQL> CREATE OR REPLACE TRIGGER after_update_email
  2  AFTER UPDATE OF Email ON User_1128
  3  FOR EACH ROW
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE('Email address updated for user: ' || :OLD.Name);
  6  END;
  7  /

Trigger created.
```

6. Create an AFTER DELETE Trigger for the “User” Table that signals when a user is deleted.

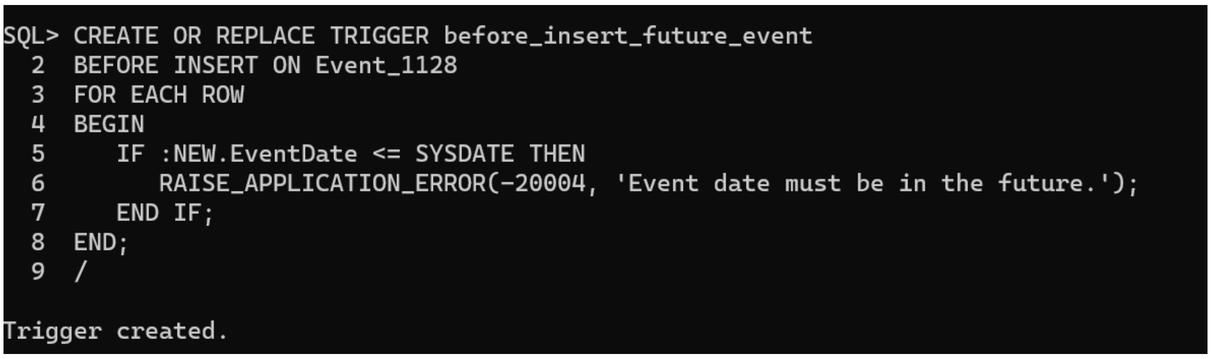
```
CREATE OR REPLACE TRIGGER after_delete_user_signal
AFTER DELETE ON User_1128
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('User deleted: ' || :OLD.Name);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_delete_user_signal
  2  AFTER DELETE ON User_1128
  3  FOR EACH ROW
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE('User deleted: ' || :OLD.Name);
  6  END;
  7  /

Trigger created.
```

7. Create a BEFORE INSERT Trigger for the "Event" Table that ensures the event's date is in the future.

```
CREATE OR REPLACE TRIGGER before_insert_future_event
BEFORE INSERT ON Event_1128
FOR EACH ROW
BEGIN
  IF :NEW.EventDate <= SYSDATE THEN
    RAISE_APPLICATION_ERROR(-20004, 'Event date must be in the future.');
  END IF;
END;
/
```

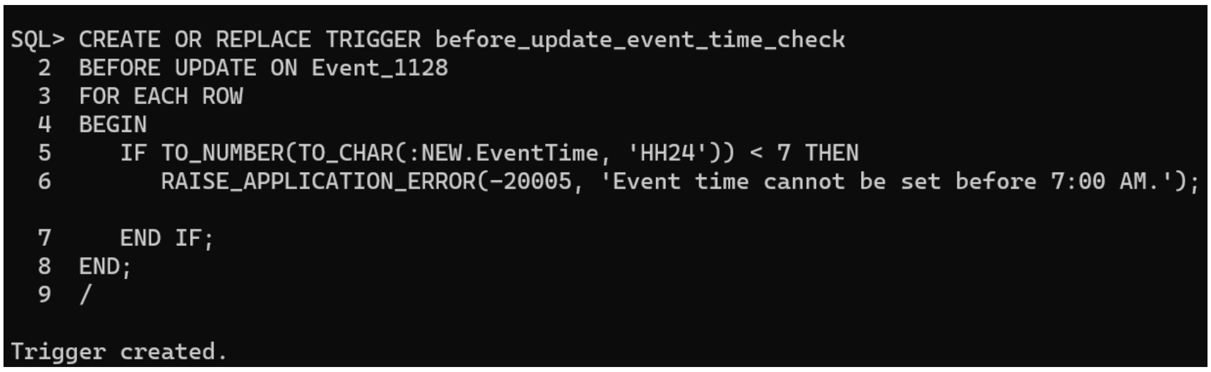


```
SQL> CREATE OR REPLACE TRIGGER before_insert_future_event
  2  BEFORE INSERT ON Event_1128
  3  FOR EACH ROW
  4  BEGIN
  5     IF :NEW.EventDate <= SYSDATE THEN
  6        RAISE_APPLICATION_ERROR(-20004, 'Event date must be in the future.');
  7     END IF;
  8  END;
  9  /

Trigger created.
```

8. Create a BEFORE UPDATE Trigger for the "Event" Table that Ensures that the event's time is not set to before 7:00 AM (assuming you use 24-hour format for your Time column).

```
CREATE OR REPLACE TRIGGER before_update_event_time_check
BEFORE UPDATE ON Event_1128
FOR EACH ROW
BEGIN
  IF TO_NUMBER(TO_CHAR(:NEW.EventTime, 'HH24')) < 7 THEN
    RAISE_APPLICATION_ERROR(-20005, 'Event time cannot be set before 7:00
AM.');
  END IF;
END;
```

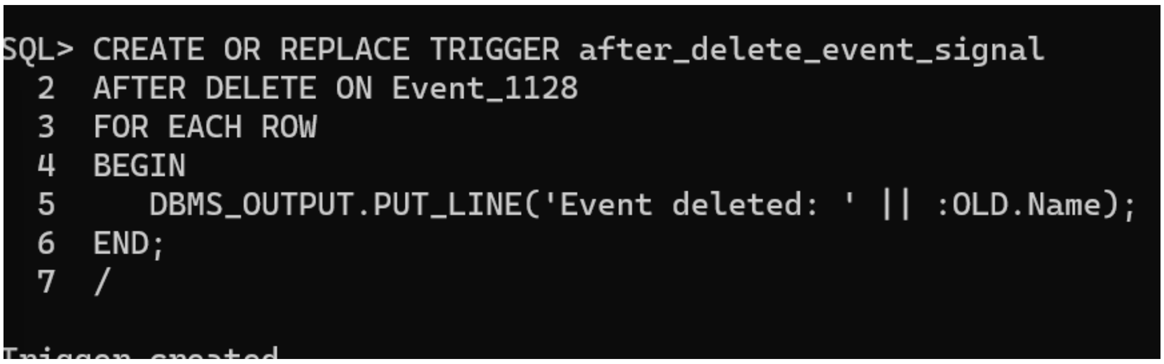
/



```
SQL> CREATE OR REPLACE TRIGGER before_update_event_time_check
  BEFORE UPDATE ON Event_1128
  FOR EACH ROW
  BEGIN
     IF TO_NUMBER(TO_CHAR(:NEW.EventTime, 'HH24')) < 7 THEN
        RAISE_APPLICATION_ERROR(-20005, 'Event time cannot be set before 7:00 AM.');

     END IF;
  END;
  /

Trigger created.
```

9. Create an AFTER DELETE Trigger for the "Event" Table that signals when an event is deleted.

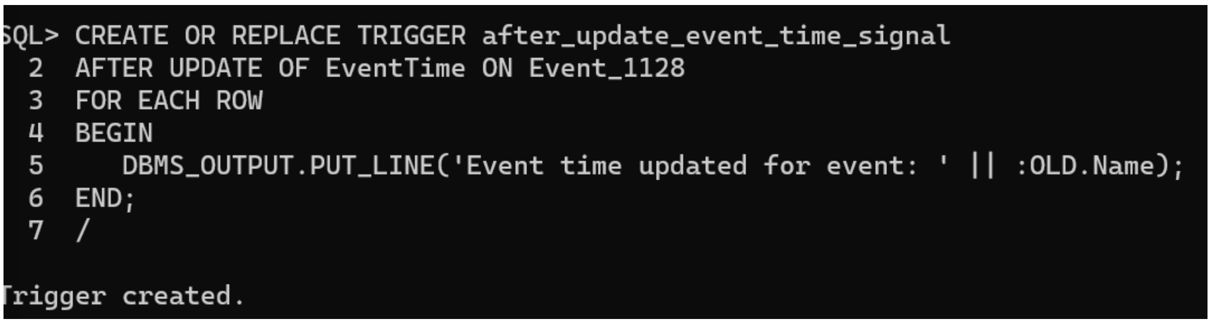
```
CREATE OR REPLACE TRIGGER after_delete_event_signal
AFTER DELETE ON Event_1128
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Event deleted: ' || :OLD.Name);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_delete_event_signal
  AFTER DELETE ON Event_1128
  FOR EACH ROW
  BEGIN
     DBMS_OUTPUT.PUT_LINE('Event deleted: ' || :OLD.Name);
  END;
  /

Trigger created.
```

10. Create an AFTER UPDATE Trigger for the "Event" Table that signals when an event's time is changed.

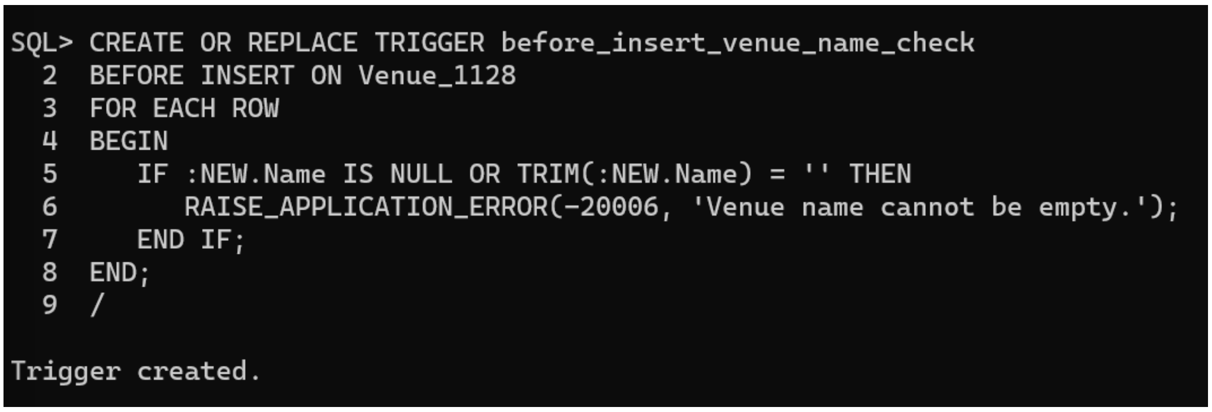
```
CREATE OR REPLACE TRIGGER after_update_event_time_signal
AFTER UPDATE OF EventTime ON Event_1128
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Event time updated for event: ' || :OLD.Name);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_update_event_time_signal
  AFTER UPDATE OF EventTime ON Event_1128
  FOR EACH ROW
  BEGIN
     DBMS_OUTPUT.PUT_LINE('Event time updated for event: ' || :OLD.Name);
  END;
  /

Trigger created.
```

11. Create a BEFORE INSERT Trigger for the "Venue" Table that ensures the name of the venue is not empty.

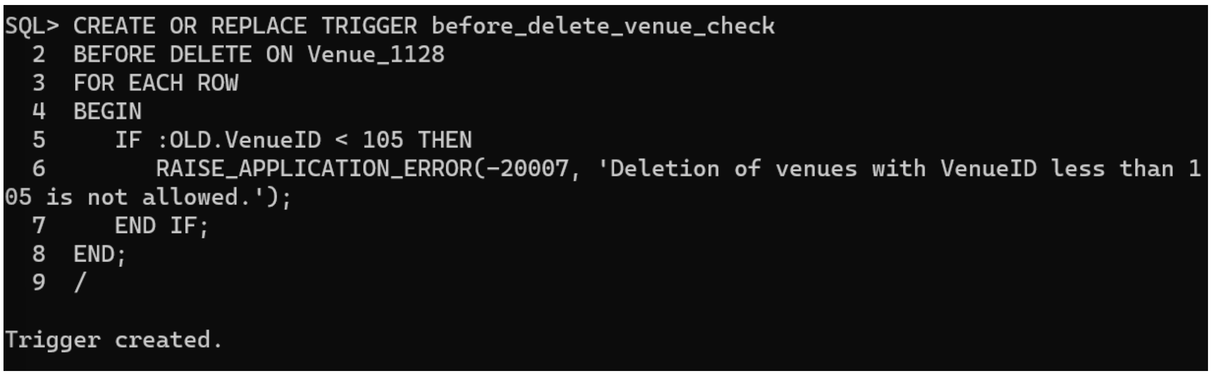
```
CREATE OR REPLACE TRIGGER before_insert_venue_name_check
BEFORE INSERT ON Venue_1128
FOR EACH ROW
BEGIN
   IF :NEW.Name IS NULL OR TRIM(:NEW.Name) = '' THEN
      RAISE_APPLICATION_ERROR(-20006, 'Venue name cannot be empty.');
   END IF;
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER before_insert_venue_name_check
  2  BEFORE INSERT ON Venue_1128
  3  FOR EACH ROW
  4  BEGIN
  5     IF :NEW.Name IS NULL OR TRIM(:NEW.Name) = '' THEN
  6        RAISE_APPLICATION_ERROR(-20006, 'Venue name cannot be empty.');
  7     END IF;
  8  END;
  9  /

Trigger created.
```

12. Create a BEFORE DELETE Trigger for the “Venue” Table that Prevents deletion if the VenueID is less than 105.

```
CREATE OR REPLACE TRIGGER before_delete_venue_check
BEFORE DELETE ON Venue_1128
FOR EACH ROW
BEGIN
   IF :OLD.VenueID < 105 THEN
      RAISE_APPLICATION_ERROR(-20007, 'Deletion of venues with VenueID less than
105 is not allowed.');
   END IF;
END;
/
```

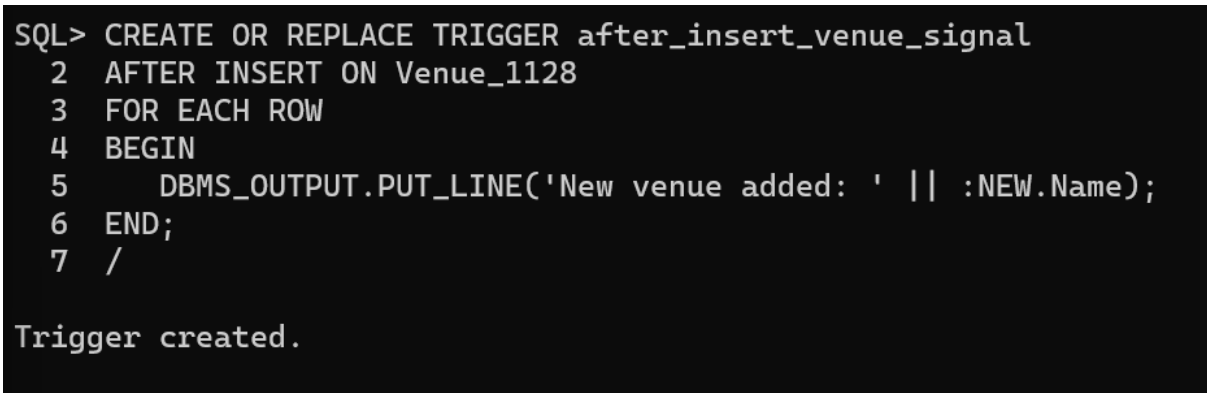


```
SQL> CREATE OR REPLACE TRIGGER before_delete_venue_check
  2  BEFORE DELETE ON Venue_1128
  3  FOR EACH ROW
  4  BEGIN
  5     IF :OLD.VenueID < 105 THEN
  6        RAISE_APPLICATION_ERROR(-20007, 'Deletion of venues with VenueID less than 1
05 is not allowed.');
  7     END IF;
  8  END;
  9  /

Trigger created.
```

13. Create an AFTER INSERT Trigger for the “Venue” Table that signals when a new row is added to it.

```
CREATE OR REPLACE TRIGGER after_insert_venue_signal
AFTER INSERT ON Venue_1128
FOR EACH ROW
```

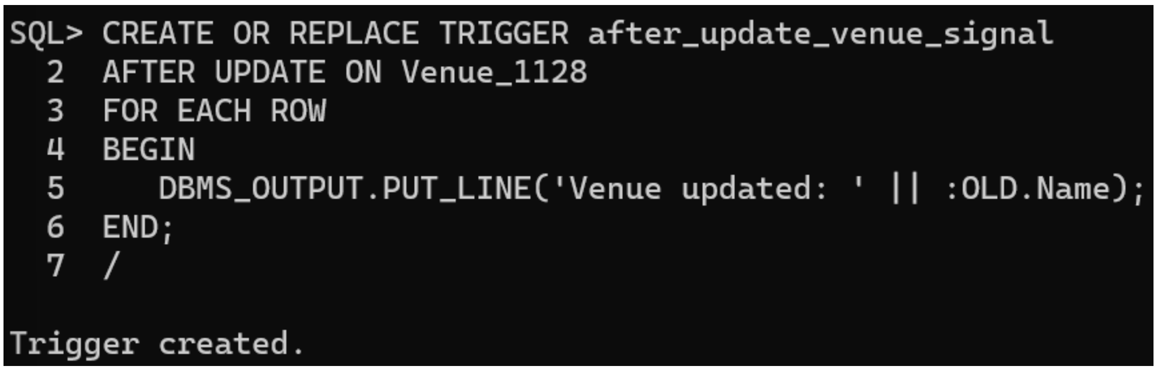
```
BEGIN
   DBMS_OUTPUT.PUT_LINE('New venue added: ' || :NEW.Name);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_insert_venue_signal
  2  AFTER INSERT ON Venue_1128
  3  FOR EACH ROW
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE('New venue added: ' || :NEW.Name);
  6  END;
  7  /

Trigger created.
```

14. Create an AFTER UPDATE Trigger for the "Venue" Table that signals when a row is updated.

```
CREATE OR REPLACE TRIGGER after_update_venue_signal
AFTER UPDATE ON Venue_1128
FOR EACH ROW
BEGIN
   DBMS_OUTPUT.PUT_LINE('Venue updated: ' || :OLD.Name);
END;
/
```



```
SQL> CREATE OR REPLACE TRIGGER after_update_venue_signal
  2  AFTER UPDATE ON Venue_1128
  3  FOR EACH ROW
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE('Venue updated: ' || :OLD.Name);
  6  END;
  7  /

Trigger created.
```

Result:

The given trigger were created successfully.